Product Feature Prioritization Using the Hidden Structure Method: A Practical Case at Ericsson

Robert Lagerström[1], Mattin Addibpour[2], Franz Heiser[2]
[1]KTH Royal Institute of Technology, Stockholm, Sweden
[2]Ericsson AB, Stockholm, Sweden

Abstract--**In this paper, we present a case were we employ the Hidden Structure method to product feature prioritization at Ericsson. The method extends the more common Design Structure Matrix (DSM) approach that has been used in technology management (e.g. project management and systems engineering) for quite some time in order to model complex systems and processes. The hidden structure method focuses on analyzing a DSM based on coupling and modularity theory, and it has been used in a number of software architecture and software portfolio cases. In previous work by the authors the method was tested on organization transformation at Ericsson, however this is the first time it has been employed in the domain of product feature prioritization. Today, at Ericsson, features are prioritized based on a business case approach where each feature is handled isolated from other features and the main focus is customer or market-based requirements. By employing the hidden structure method we show that features are heavily dependent on each other in a complex network, thus they should not be treated as isolated islands. These dependencies need to be considered when prioritizing features in order to save time and money, as well as increase end customer satisfaction.**

I. INTRODUCTION

Today, at Ericsson, features are prioritized based on a business case approach where each feature is handled isolated from other features and the main focus is customer or market-based requirements. In reality the dependency relations between features are far more complicated than the immediate business view. The development of features is not done in isolation, and many aspects are common and reused. In large-scale software development organizations, parallel development of features needs to be planned carefully taking into consideration the indirect feature dependencies through development dimensions, e.g. design base, organizational, and infrastructure. Although the legacy and code design base dependencies are somewhat included in the analysis, a more elaborated dependency analysis for optimizing the usage of development dimensions and maximizing parallel development needs to be explored. Many of these dependencies are not visible using the methods currently in place at Ericsson, which focus only on direct dependencies (first order dependencies) and not indirect dependencies that often matter too [2][16].

For analyzing the direct and indirect dependencies between product features we have applied an evolution of Design Structure Matrices (DSMs) called "Hidden Structure" [2], a methodology previously used for analysis of large software products like Linux, Mozilla, Apache, and GnuCash [2] or application portfolios [11][12] and enterprise architecture [9]. In an earlier paper, we applied the hidden structure methodology to organizational transformation [7]. In this paper we use the methodology for software feature prioritization.

By employing the hidden structure method we explore a methodology that reveals the hidden dependencies between features in a complex network. The patterns revealed in this paper, help the process of prioritization and planning, by categorizing features into four different types that require different planning strategies, e.g. features that should be planned together, those that can be developed in parallel and so forth. These dependencies need to be considered when prioritizing features in order to optimize the usage of common resources, as well as increase end customer satisfaction by making well-informed decisions in a structured way.

We show that in our practical case at Ericsson the visualization of the patterns that the hidden structure method provides, can be used to redefine the priority of the features to make an optimal feature planning strategy. Beside a more elaborated feature grouping and prioritization, several additional improvement potentials are revealed as a result of our analysis, such as; test case planning, delivery strategy, and smart team allocation. To facilitate the permanent process of planning features, we have realized a tool called ADP (Advanced DSM Processing) that supports all the needed steps to calculate and describe the hidden patterns. ADP is intented to be described in a later publication.

The paper unfolds as follows: chapter 2 presents related work and chapter 3 the hidden structure method. In chapter 4 our case is described together with its results and a discussion. Future work is outlined in chapter 5. And finally, chapter 6 concludes the paper.

II. RELATED WORK

Design Structure Matrices (DSMs) [4][14][18][21] has been used for some time to model and analyze complex systems incl. software architecture, mechanical systems, physical systems, and organizations [3]. Eppinger and Browning [3] presents four different types of DSMs, namely; 1) product architecture, 2) organization architecture, 3) process architecture, and 4) multi-domain. The case presented in this paper mainly falls into category 1, product architecture. Further, they list that the main benefits with using DSMs for system architecture modeling are;

conciseness, visualization, intuitive understanding, analysis, and flexibility.

As an evolution of DSMs Baldwin et al. [2] developed a method to visualize the hidden structure of software architectures based classic coupling measures. This method has been tested on numerous software products, such as Linux, Mozilla, Apache, and GnuCash. In one study by MacCormack et al. [15] an early version of this hidden structure method was employed to show the relation between the product architecture and its development organization. Others have used the DSM approach and also hidden structure-based metrics for various purposes [13][17][22].

Recently the fields of enterprise architecture [23] and software portfolio management started to test the hidden structure method with the aim to visualize and measure these types of systems. In [9] the hidden structure method was used on a Biopharmaceutical case to reveal the hidden dependencies in its enterprise architecture incl. business groups, software applications, databases, schemas etc. Data from this case was then used to show that the cost of changing applications with many indirect dependencies (metrics derived using the DSM based hidden structure method) was more expensive than applications with few [10][16]. Also in [11] the authors employed the hidden structure method on application portfolio data from a Telecommunication case and in [12] from a power utility case. Another very highly related study using the hidden structure method is the one presented in [7], where the authors employed the method in order to reveal hidden structures in organizational transformation using a case at Swedish telecom company Ericsson.

All in all these case studies build up a story of interesting application areas for DSMs and hidden structure. However, the focus in the hidden structure papers have either been on software code bases or enterprise application portfolios. In this paper we instead aim to make practical use of the DSM approach using the hidden structure method for software product feature prioritization. It is thus a new application area for the method.

III. HIDDEN STRUCTURE

The method we use for representing software product features and their interactions is based on and extends the classic notion of coupling [5][6][8][20] and modularity [1][19]. Specifically, after identifying the dependencies (coupling) between the features, we analyze them as a system in terms of hierarchical ordering and cyclic groups, and classify features in terms of their position in the resulting network/system (this method is more thoroughly described in [2].

In a Design Structure Matrix (DSM), each diagonal cell represents an element/node (for us here a feature), and the off-diagonal cells record the dependencies between the elements (links): If element i depends on element j, a mark is placed in the row of i and the column of j. The content of the matrix does not depend on the ordering of the rows and columns, but different orderings can reveal (or obscure) the underlying structure. Specifically, the elements in the DSM can be arranged in a way that reflects hierarchy, and, if this is done, dependencies that remain above the main diagonal will indicate the presence of cyclic interdependencies (A depends on B, and B depends on A). The rearranged DSM can thus reveal significant facts about the underlying structure of the architecture that cannot be inferred from standard measures of coupling. In the following subsections, a method that makes this "hidden structure" visible is presented.

A. Identify the Direct Dependencies and Compute the Visibility Matrix

Any complex system can be represented as a directed graph composed of N elements (nodes) and directed dependencies (links) between them. This directed graph can be represented as a DSM. If the DSM is raised to successive powers, the result will show the direct and indirect dependencies that exist for successive path lengths. Summing these matrices yields the visibility matrix V (or VSM), the far right matrix in Fig. 1, which denotes the dependencies that exist for all possible path lengths. The values in the visibility matrix are constrained to be binary, capturing only whether a dependency exists and not the number of possible paths that the dependency can take [14]. The matrix for n=0 (i.e., a path length of zero) is included when calculating the visibility matrix, implying that a change to an element will always affect itself.

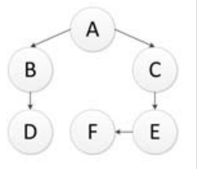
A Directed Graph	Design Structure Matrix	Visibility Matrix $V=\sum M^n$; $n=[0,4]$
	$\begin{array}{c\|cccccc} & A & B & C & D & E & F \\ A & 0 & 1 & 1 & 0 & 0 & 0 \\ B & 0 & 0 & 0 & 1 & 0 & 0 \\ C & 0 & 0 & 0 & 0 & 1 & 0 \\ D & 0 & 0 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 0 & 0 & 1 \\ F & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$	$\begin{array}{c\|cccccc} & A & B & C & D & E & F \\ A & 1 & 1 & 1 & 1 & 1 & 1 \\ B & 0 & 1 & 0 & 1 & 0 & 0 \\ C & 0 & 0 & 1 & 0 & 1 & 1 \\ D & 0 & 0 & 0 & 1 & 0 & 0 \\ E & 0 & 0 & 0 & 0 & 1 & 1 \\ F & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$

Fig. 1. A directed graph with the corresponding DSM and VSM.

Several measures are constructed based on the VSM. First, for each element i in the architecture, the following are defined:

- VFI_i (Visibility Fan-In) is the number of elements that directly or indirectly depend on i. This is found by summing entries in the i^{th} column of V.
- VFO_i (Visibility Fan-Out) is the number of elements that i directly or indirectly depends on. This is found by summing entries in the i^{th} row of V.

In Fig. 1, element A has a VFI equal to 1, meaning that no other elements depend on it, and a VFO equal to 6, meaning that it depends on all other elements in the architecture. To measure visibility at the system level, *Propagation Cost* (*PC*) is defined as the density of the VSM. Intuitively, propagation cost equals the fraction of the architecture that may be affected when a change is made to a randomly selected element.

B. Identify and Rank Cyclic Groups

The next step is to find the cyclic groups in the system. By definition, each element within a cyclic group depends directly or indirectly on every other member of the group. First, the elements are sorted, first by *VFI* descending and then by *VFO* ascending. Next one proceeds through the sorted list to find different cyclic groups. These groups are referred to as the "cores" of the system. The largest cyclic group is defined as the "Core". Once the Core is identified, the other components in the architecture can be classified into groups, as follows:

- "Core" elements are members of the largest cyclic group and have the same *VFI* and *VFO*, denoted by VFI_C and VFO_C, respectively.
- "Control" elements have $VFI < VFI_C$ and $VFO \geq VFO_C$.
- "Shared" elements have $VFI \geq VFI_C$ and $VFO < VFO_C$.
- "Periphery" elements have $VFI < VFI_C$ and $VFO < VFO_C$.

Using the above classification scheme, a reorganized DSM can be constructed that reveals the "hidden structure" of the system by placing elements in the order Shared, Core, Periphery, and Control down the main diagonal of the DSM, and then sorting within each group by *VFI* descending and then *VFO* ascending (cf. Fig. 3 for an example of a hidden structure sorted DSM).

The method for classifying systems into different types of architectures is discussed in empirical work by Baldwin et al. in [2]. Specifically, the authors find a large percentage of the systems they analyzed contained a large cyclic group of components that was dominant in two senses: i) it was large relative to the number of elements in the system, and ii) it was substantially larger than any other cyclic group. This architectural type is classified as "core-periphery." Where architectures have multiple cyclic groups of similar size, the architecture is referred to as "Multi-Core". Finally, if the Core is small, relative to the system as a whole, the architecture is referred to as "Hierarchical."

IV. CASE

In this section we describe how the Hidden Structure method, an evolution of the Design Structure Matrix (DSM) approach, was applied to a case at Ericsson AB in order to reveal unknown structures between software product features.

The main focus in this paper is the interaction between a number of software (SW) features to be implemented in LTE Radio Access Networks delivered by Ericsson, owned by Mobile Operators like e.g. AT&T in the US or CMCC in China. The SW features are intended to increase the user experience, network capacity, and/or the network performance of mobile terminals (UE) in a Radio Access Network consisting of numerous Radio Base Stations (RBS), deployed in-house or outdoors. The Radio Base Stations can serve GSM, WCDMA or LTE Access Technologies. We focus on LTE features with emphasis on Mobile Broadband (MBB). MBB is a term describing the overall categorization of services available for UEs. Mobile Broadband services can include e.g.;

- Voice over LTE (VoLTE),
- File downloads,
- HTTP(S) sessions like Youtube, etc.,
- Broadcast services, and
- APP downloads.

Examples for improvements of user experience are:

- Increase of the achievable UE download speed.
- Better VoLTE coverage at cell edges.
- Better VoLTE quality and less experience of latency to the UE of a mobile subscriber.
- Less buffering time for Youtube sessions until video start.
- Higher resolution for Youtube video sessions.

Examples for Radio Network Performance & Capacity are:

- More VoLTE users per cell, RBS in a whole network.
- More simultaneous video downloads at high resolution.
- Increased number of Broadcast channels available for mobile subscribers in a Radio Network.

Improvements of user experience, network performance, and/or network capacity are broken down into SW features, which the SW design teams then are implementing.

The delivery of SW features is planned for six month Main Release Periods. A Year is split into an "A" and "B" release, e.g. L14A, L14B, L15B. The "L" stands for LTE, "A" Release is before "B" release, i.e. L14A is delivered to customers before L14B release. During the analysis we have looked at three different areas with two different releases in each, however for this paper we have selected a release for Mobile Broadband (MBB) features delivered during 2015 in the "B" release (L15B). In this release a number of SW features were realized. Examples for features analyzed in this paper are:

- Carrier aggregation between FDD and TDD.
- Downlink Carrier Aggregation across multiple frequencies.
- Quad antenna configurations in RBS.
- Coordinated scheduling and link adaptation in an RBS.
- Increased number of cells per RBS.

We are not revealing further details of the features, as this additional information is not needed for the understanding and the conclusions drawn from our analysis. A total of 52 MBB SW features are included in the analysis of this paper.

A Release also contains features in other areas. The features of other areas only have minor interactions with the features analyzed in this paper, therefore they are not included here.

Not all 52 examined features are intended for the L15B release, 32 were intended for the L15B release, 20 features were delivered in an earlier release.

The feature dependencies are input to the DSM as shown below in Fig. 2.

A feature interaction can be classified into three areas;

1. Functional Dependency,
2. Interaction Dependency, and
3. Code Proximity/Resource Dependency.

A functional dependency is shown if the two features are functionally connected. Interaction dependency is shown if the features interact with each other, e.g. that only one can be enabled at a time, or both need to be enabled at the same time. Code proximity/Resource Dependency is shown if the features affect the same HW resource or require code changes in the same SW unit or file.

The reason why we also include previous features in our analysis is mainly because new features are not only depending on other new features, but also on legacy features via the same above described feature interaction classes.

Although the 32 features are part of the L15B Release, they are individually prioritized among all features intended for the release. The prioritization process is strictly business oriented, i.e. the feature with best business case has highest ranking. The ranking is indicated by an increasing number, whereof increasing number means less priority and the lowest number thus has highest priority. Each feature has a corresponding priority number. Priority numbers for MM L15B release are not strictly sequential due to Ericsson internal reasons. Earlier features have therefore no priority number in the 15B release analysis, as they have been already implemented.

A. Input Data

Each MBB SW feature is indicated by a feature number (1-52). Fig. 2 below shows the input for the first order feature dependency matrix (the input DSM). Features 1-32 below are the 15B features, features 33-52 are legacy features, implemented in earlier releases.

Item	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52
1	x															x																																				
2		x				x										x				x															x			x					x									
3			x			x												x		x															x		x	x					x									
4				x														x																			x															
5					x																																															
6		x				x								x			x			x																							x									
7		x					x				x		x	x			x			x															x														x			
8								x			x		x																						x											x	x		x	x		
9									x																										x											x				x		
10										x																																										
11											x			x	x		x			x				x							x			x	x		x	x	x											x	x	x
12					x							x		x	x		x			x				x						x	x			x	x		x	x	x								x			x	x	x
13					x			x			x	x	x	x	x		x			x				x						x	x			x	x		x	x	x							x	x	x	x	x	x	
14								x			x	x	x	x	x		x			x				x						x	x		x	x	x		x	x	x				x			x	x	x	x	x	x	
15											x		x		x		x																x		x	x							x									
16											x		x			x	x																x		x	x																
17								x			x		x				x											x		x	x						x		x		x					x		x	x	x		
18								x			x		x					x										x		x	x						x		x		x					x		x	x	x		
19																			x				x												x		x			x		x										
20		x				x					x		x	x						x			x	x											x		x	x		x		x	x									
21		x				x					x		x	x							x			x											x			x			x		x					x				
22																						x																														
23																		x					x		x						x	x																				
24											x		x					x		x				x	x						x	x			x			x											x	x		
25		x									x		x							x					x										x			x											x	x		
26																										x			x																							
27																											x		x																							
28																												x																								
29																													x																							
30																														x																						
31																															x																					
32																																x																				
33																																	x																			
34																																		x																		
35																																			x																	
36																																				x																
37																																					x															
38																																						x														
39																																							x													
40																																								x												
41																																									x											
42																																										x										
43																																											x									
44																																												x								
45																																													x							
46																																														x						
47																																															x					
48																																																x				
49																																																	x			
50																																																		x		
51																																																			x	
52											x	x																																								x

Fig. 2. "MBB_L15B" input DSM.

B. Problem Description

Feature dependencies in first order have been used at Ericsson for quite some time. The reason for this is that feature dependencies can help understand on beforehand the following problem areas:

- Feature teams destroy each other's code changes.
- Features already delivered to customers suddenly stopped working.
- Wrong planning due to missing understanding and consequences of feature dependencies.
- Inappropriate test setups which were missing feature dependency, resulting in bad quality.
- Late started features destroyed code base of earlier started features, on the way delivered to customers.

Although feature interaction tables were created, the above problems improved, but did not really disappear. Several reasons might be possible, like e.g.:

- Not all feature dependencies were included, hidden dependencies might be important.
- Features are started in wrong order, not reflecting the dependency direction between them.
- Corrections for Trouble Reports (TR) from operators on legacy features might affect new features.
- Large features with many dependencies implemented by feature teams, not capable to communicate their changes to other teams implementing dependent features.

The hidden structure method as outlined in chapter 3 is used in order to help reveal reasons causing some of these problems.

C. Use of the Method and Results

Fig. 3 shows the DSM sorted according to the Hidden Structure method. The analysis revealed a Core-Periphery architecture. The cyclic groups are placed along the diagonal from top left to bottom right in the usual order; Shared, Core, Periphery, and Control.

Shared Group:

The Shared group contains only legacy features except feature no 5, which is a common feature across not only MBB but also other areas. Interestingly it is also in the shared group in the MBB feature set. Legacy features experience corrections and changes due to Trouble Reports (TR) from operators. The Priority is not MBB Controlled. TR's in these features might destroy baseline/legacy for Core / Control Group features. TR correction & feature implementation needs highest prio/earliest delivery time (from MBB perspective).

Core Group:

We found that feature priority is rather low in average (high prio numbers). TR's in these features might destroy baseline/legacy for features belonging to the cyclic group for control.

Periphery Group:

This group can be seen as a "plan-as-you-like-when-it-fits" group. Many features in this group can increase planning flexibility. Unfortunatelly only a small amount of features (8) belong to this group.

Control Group:

Business priority is rather high in average (low prio numbers). TR's or new code from Core Group having mainly low business priority very likely to destroy already performed verification in this group with high business priority.

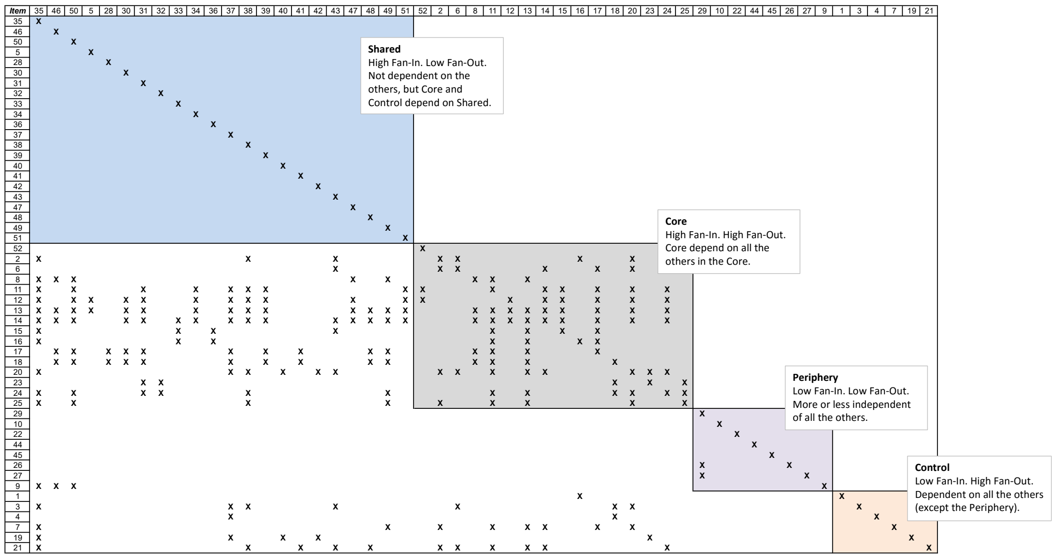

Item
35 46 50 5 28 30 31 32 33 34 36 37 38 39 40 41 42 43 47 48 49 51 52 2 6 8 11 12 13 14 15 16 17 18 20 23 24 25 29 10 22 44 45 26 27 9 1 3 4 7 19 21
Shared
High Fan-In. Low Fan-Out. Not dependent on the others, but Core and Control depend on Shared.
Core
High Fan-In. High Fan-Out. Core depend on all the others in the Core.
Periphery
Low Fan-In. Low Fan-Out. More or less independent of all the others.
Control
Low Fan-In. High Fan-Out. Dependent on all the others (except the Periphery).

Fig. 3. "MBB_L15B" restructured DSM based on the hidden structure classification.

D. Interpretation

Fig. 3 was input to an analysis of the prioritization between cyclic groups. As mentioned earlier, Peripheral features can be planned independently due to their independence of all the others. Therefore this group is not visible in the analysis about prioritization.

Table 1 shows the priority numbers for individual features in the different cyclic groups.

TABLE 1. PRIORITY NUMBERS FOR INDIVIDUAL FEATURES IN THE DIFFERENT CYCLIC GROUPS.

Feature Priority Number		
Shared	*Control*	*Core*
31	22	28
	29	32
	30	34
	33	37
	45	38
	47	39
		40
		41
		42
		43
		44
		46
		49
		50
		52

One feature in L15B is categorized as Shared and had priority number 31. Median and Mean of the priority number of the Control Group is 33 and 34, respectively. For the Core Group the Median and Mean priority values are both 41. This means that Core features are lying approximately 7-8 priority numbers above Core. This seems very low, but if we look on a bar chart showing the distribution, then the difference between the groups becomes more evident, see Fig. 4.

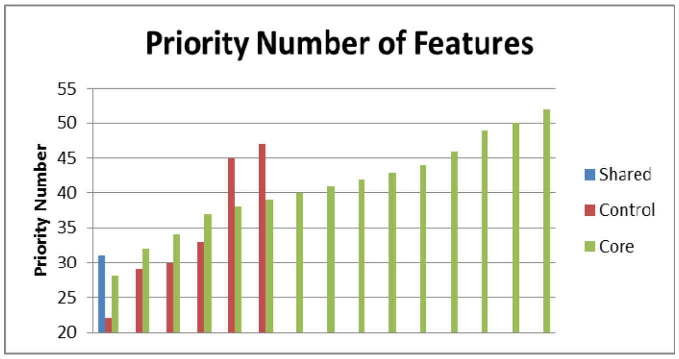


Fig. 4. Priority number of features in Shared, Control, and Core groups.

The following can be derived from Fig 4:

- Only two out of six Control features have a very high priority number, i.e. low priority. Four Control feature have very high priority, i.e. very low priority numbers.
- 13 out of 15 Core features have lower priority than the 4 Core features with highest priority.
- The Shared feature in L15B has the same priority as the Control features.

Control Group features have highest business value and are therefore highly prioritized, started early, and verified early. Control group features are dependent on Core and Shared group features; implementing high priority features in the Control group therefore results in an early start of Control group features. Later addition of Shared and Core group features destroy stability and invalidate the earlier already done testing efforts.

By doing this, the high priority features are at most risk to get broken, when the SW updates including late additions of low priority content are designed and delivered. Trouble reports from field changing the Shared cyclic group might impact the Control featureas also in a negative way. The correction of a Trouble Report (TR) from previous releases will create new TRs in the Control group features due to their dependencies to features in the Shared group.

In order to avoid the risk that Control features are broken, when Core and Shared feature are started or due to TRs changing the Shared group, a number of countermeasures can be considered:

- Shared and Core group features and TRs must be finished and verified before Control group features can be delivered and verified.
  - o If not done: Unpredictable baseline for Control group features, which have highest business priority and highest customer focus.
- Every new feature and TR correction in the Shared group should be coordinated with Core and Control group features for impact analysis.
  - o If not done: Unpredictable baseline for all Core and Control features.
- Fast TR correction time with high priority for features in Shared group.
  - o If not done: Instable baseline for all Core and Control features.

Other possible considerations:

- Wrong order / priority can make it impossible to deliver high priority features in Control group with acceptable quality with an inherent risk for customer escalations.
- As Control group features are dependent on all other features, root causes for Control group TRs are almost impossible to find with reasonable effort if Shared and Core features are unstable or badly verified.

E. Discussion

We have shown that DSM analysis with the Hidden Structure method can be used to impact the priority of SW features. Careful planning with the right prioritization of the cyclic groups is essential to prohibit the problems as outlined in the problem description above. Shared features/TRs should have highest priority, followed by Core group features having second highest priority, and then the Control group features. Periphery features can be planned independently.

Besides the prioritization between features, several additional improvements in SW design can be seen for each cyclic feature group. We describe such ideas only briefly here:

- Feature dependencies should be considered in test case analysis, test case implementation, and test case execution.
- Feature dependencies should be identified as early as possible in the development phases (system design phase), to have an early version of the DSM as input to later phases.
- Feature dependencies should steer the delivery strategy, both internally and externally to customers.
- Relate team competence to cyclic groups.
  - Shared feature teams should have high communication skills in order to communicate changes to dependent Core and Control features.
  - Core feature teams should be highly competent to collaborate.
  - Control feature teams should be highly competent to find “others” mistakes.

Other improvement possibilities, sorted per cyclic group is also possible like e.g.:

Peripheral features:

- Reduce test case scope for legacy testing compared to other cyclic feature groups, due to the limited number of dependencies.
- Assess the possibility to skip legacy testing completely for this group.

Shared features:

- Define higher quality levels to be achieved before a shared feature/TR can be delivered internally or externally to customer, minimizing the risk for broken Core or Control features.

Core features:

- Think and enable smart testing by coordinated testing in order to take advantage of the common dependency structure.
- Use the same team for several features to speedup startup time, as the dependencies within the Core feature group are identical.
- Re-use test environment within the group, take advantage of the dependency structure.
- For late Core features - an improved test analysis is needed due to Control feature dependencies. The creation of support teams to help Core feature teams not to break Control features will improve SW quality.

Control features:

- For early Control features - differentiate the interaction with Shared and Core features, make the dependencies visible to all feature teams that the Control features are dependent on.

As the planning of features is a permanent process, it is important that the DSM created and the Hidden Structure analyzed in very timely manner. We have therefore realized a tool for DSM analysis using the hidden dependency method. The tool is able to import several input formats for the DSM and then run the Hidden Structure method within seconds. The tool can also compare different versions of a DSM and VSM, and changes can be highlighted for impact analysis. The tool is called ADP (Advanced DSM Processing) and will be described in a later publication.

V. FUTURE WORK

In this paper we present a case for feature dependencies analysis based on function and interaction dependencies between Ericsson software features, including the dependencies originating from the code base. It would be very interesting to also include other development dimensions such as organization, processes, and development artifacts (e.g. tools, production related code, information, and machines) in the analysis. We are currently experimenting with so called multi-domain hidden structure analysis that will pave the way for a more advanced analysis of feature dependencies. The main challenge here is to “normalize” the values and priorities between different artifact types, since the type of relations in each domain has certain semantics that make perfect sense within a domain, but could be difficult to interpret between different domains.

In our model all dependencies are binary; this results in a somewhat worst-case analysis and not a weighted or most probable one. In order to make the analysis more precise the dependencies between the elements could be weighted. A weighted approach can be even more complex if it is applied in a multi-domain case. Another interesting approach is to consider N-1 order dependencies in relation to the N^{th}-order (as currently used in the Hidden Structure method), since the dependency relations tend to become weaker and less probable for each step closer to N.

By continuous development of our analysis tool (called ADP), we find it very easy to apply the methodology to different domains and applications at Ericsson AB. We continuously find new areas where the method is applicable.

On such new area we are currently experimenting with is a value-based analysis of improvement proposals. The idea is to apply the methodology in order to select the best improvement proposals out of many suggested, by analyzing the dependencies between them and maximizing the delivered value. We consider this approach as a complement to the currently used business case analysis. Another area we intend to explore is to help proposing simplified first order dependencies in our software structure by reducing the total numner of hidden dependencies (i.e. propagation cost) in the hidden structure (i.e. VSM).

VI. CONCLUSIONS

In this paper we test the Hidden Structure method, a dependency analysis approach based on Design Structure Matrix (DSM) input, on a software feature prioritization case at Ericsson. The method provides valuable input to the feature prioritization strategy and acts as a complement to the business case analysis currently setting the implemention order of features. The different business groups testing the method at Ericsson all agree that the approach works great for this application domain (software feature dependency analysis). It provides valuable information currently being missed in the analysis. People found it intuitive to work with and the tool support developed within Ericsson helped spreading the method to a larger internal audience.

REFERENCES

[1] Baldwin, C. and K. Clark, *Design Rules - Volume 1: The Power of Modularity*. MIT Press, 2000.

[2] Baldwin, C., A. MacCormack and J. Rusnak, “Hidden structure: Using network methods to map system architecture,” *Research Policy*, vol. 43, no. 8, pp. 1381-1397, 2014.

[3] Eppinger, S. and T. Browning, *Design Structure Matrix Methods and Applications*. The MIT Press, Cambridge MA, 2012.

[4] Eppinger, S. D., D. E. Whitney, R. P. Smith and D. A. Gebala, “A model-based method for organizing tasks in product development,” *Research in Engineering Design*, vol. 6, no. 1, pp. 1-13, 1994.

[5] Fenton, N. and A. Melton, “Deriving structurally based software measures,” *Journal of Systems and Software,* vol. 12, no. 3, pp. 177–187, 1990.

[6] Hall, N. R. and S. Preiser, “Combined network complexity measures,” *IBM journal of research and development,* vol. 28, no. 1, pp. 15-27, 1984.

[7] Heiser, F., R. Lagerström and M. Addibpour, “Revealing hidden structures in organizational transformation: A case study,” in *Trends in Enterprise Architecture Research (TEAR)* workshop, Springer, 2015.

[8] Henry, S. and D. Kafura, “Software structure metrics based on information flow,” *IEEE Transactions on Software Engineering,* vol. 5, pp. 510-518, 1981.

[9] Lagerström, R., C. Baldwin, A. MacCormack and D. Dreyfus, “Visualizing and measuring enterprise architecture: An exploratory BioPharma case,” in *the 6th IFIP WG 8.1 Working Conference on the Practice of Enterprise Modeling (PoEM)*, Springer, 2013.

[10] Lagerström, R., C. Baldwin, A. MacCormack and D. Dreyfus, “Visualizing and measuring software portfolio architecture: A flexibility analysis,” in *the 16th International Dependency and Structure Modelling (DSM) conference,* 2014.

[11] Lagerström, R., C. Baldwin, A. MacCormack and S. Aier, “Visualizing and measuring enterprise application architecture: An exploratory Telecom case,” in *the 47th Hawaii International Conference on System Sciences (HICSS),* pp. 3847-3856, IEEE, 2014.

[12] Lagerström, R., C. Baldwin and A. MacCormack, “Visualizing and measuring software portfolio architecture: A Power Utility case,” *Journal of Modern Project Management*, vol. 03, no. 02, 2015.

[13] LaMantia, M., Y. Cai, A. MacCormack and J. Rusnak, “Analyzing the evolution of large-scale software systems using design structure matrices and design rule theory: Two exploratory cases,” in *the 7th Working IEEE/IFIP Conference on Software Architectures (WICSA7),* 2008.

[14] MacCormack, A.; “The architecture of complex systems: Do “core-periphery" structures dominate?,” in *Academy of Management,* 2010.

[15] MacCormack, A., C. Baldwin and J. Rusnak, “Exploring the duality between product and organizational architectures: A test of the "mirroring" hypothesis,” *Research Policy,* vol. 41, no. 8, pp. 1309-1324, 2012.

[16] MacCormack, A., R. Lagerström and C. Baldwin, “A methodology for operationalizing enterprise architecture and evaluating enterprise IT flexibility,” *Harvard Business School Working Paper*, no. 15-060, 2015.

[17] MacCormack, A., J. Rusnak and C. Baldwin, “Exploring the structure of complex software designs: an empirical study of open source and proprietary code,” *Management Science,* vol. 52, no. 7, pp. 1015–1030, 2006.

[18] Sosa, M., S. Eppinger and C. Rowles, “A network approach to define modularity of components in complex products,” *Transactions of the ASME,* vol. 129, pp. 1118-1129, 2007.

[19] Parnas, D. L.; “On the criteria to be used in decomposing systems into modules,” *Communications of the ACM*, vol. 15, no. 12, pp. 1053-1058, 1972.

[20] Stevens, W. P., G. J. Myers and L. L. Constantine, “Structured design,” *IBM Systems Journal,* vol. 13, no. 2, pp. 115–139, 1974.

[21] Steward, D.; “The design structure system: A method for managing the design of complex systems,” *IEEE Transactions on Engineering Management*, vol. 3, pp. 71-74, 1981.

[22] Sturtevant, D. J.; *System design and the cost of architectural complexity.* Doctoral dissertation, Massachusetts Institute of Technology (MIT), 2013.

[23] Vakkuri, E. T.; *Developing Enterprise Architecture with the Design Structure Matrix.* Master Thesis. Tampere University of Technology, Finland, 2013.